Toward Intelligent Software Defect Detection

Learning Software Defects by Example

Markland J. Benson
Ground Software Systems Branch
Software Engineering Division
NASA Goddard Space Flight Center
Greenbelt, MD
Markland.J.Benson@nasa.gov

***Abstract*—Source code level software defect detection has gone from state of the art to a software engineering best practice. Automated code analysis tools streamline many of the aspects of formal code inspections but have the drawback of being difficult to construct and either prone to false positives or severely limited in the set of defects that can be detected. Machine learning technology provides the promise of learning software defects by example, easing construction of detectors and broadening the range of defects that can be found. Pinpointing software defects with the same level of granularity as prominent source code analysis tools distinguishes this research from past efforts, which focused on analyzing software engineering metrics data with granularity limited to that of a particular function rather than a line of code.**

Keywords: software defects, machine learning

I. Introduction

Automated source code analysis tools have matured to become industry standard in use. Popular compilers such as Microsoft Visual Studio[1], GCC[2] and *javac*[3] include options to perform what may be considered today more routine checks such as reporting uninitialized variables or unchecked type conversions. Standalone commercial products such as Polyspace[4] and Klocwork[5] detect numerous common run-time issues such as memory leaks, bad references and misuse of well-known interfaces.

Static source code analysis included in industry tools depends on rigorous control and data flow analysis. Such analyses depend on compiler construction techniques of lexing and parsing as precursors. Patterns matched in source code analyzers are hand-coded in high-level languages, as regular expressions or as state machines. Adding new rules require humans to identify specific instances of code that are considered defective, generalize a pattern that captures the defect, and encode the pattern into high-level language code, regular expressions or state machines. This research aims to take the work of generalizing and encoding defect patterns out of the realm of manual work by humans and into the realm of automated machine learning.

The remainder of this paper is structured as follows. Section II outlines a defect detector framework giving descriptions of each stage of the framework as well as some practical concerns that exist at each stage. Section III discusses work done in the area of feature extraction for translating source code into forms usable by machine learners. Section IV considers techniques applicable to classification of portions of source code as defective or non-defective. Section V looks at current work in the area of automated source code defect detection that serves to move to the next level of granularity proposed by this research. Section VI paints a picture of the next steps in achieving the goals laid out for this work.

This work is funded by the National Aeronautics and Space Administration (NASA) Goddard Space Flight Center (GSFC) Part-Time Graduate Study Program (PTGSP).

II. Defect Detector Framework

A. Framework Overview

A general pattern recognition approach is to perform preprocessing, feature extraction, classification and post-processing as is illustrated in Figure 1.

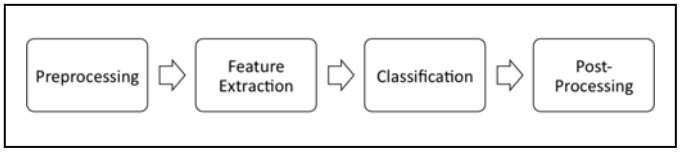

Figure 1: Pattern Recognition Pipeline

The following sections will discuss how each of these stages applies to the problem of intelligent software defect detection.

B. Preprocessing

Preprocessing shapes data into a form more usable by the classification engine. If the input data is an image, preprocessing may include translating or rotating an image to place it in a standard position and orientation or sharpening of the image to simplify the feature selection processing. In other applications where vectors or records of data are inputs, the preprocessing step may filter out inputs based on some *a priori* criteria or a statistical property of the overall dataset. Further, preprocessing may fill in missing data elements or normalizing numeric data.

While software engineering data is often in the form of metrics such as source lines of code (SLOC), defect count per function or per SLOC, cyclomatic complexity, or number of operators and operands different data is needed in order to make inferences about the presence or absence of specific types of defects for any given line of code. In this case, the recognition engine needs to ingest the source code in some less

abstract fashion than metrics. Simple approaches might use individual tokens or sets of tokens. More complex approaches would follow the control or data flow of the program.

C. Feature Extraction

Feature extraction transforms preprocessed data into a form usable by the pattern recognition engine. Creating a form that is optimized to a given machine-learning algorithm is fundamental to the application of such technology to software engineering data[6]. Pattern recognition algorithms are quite sensitive to the form of data provided them. Perceptrons, which are very simple neural networks, only handle binary 0 or 1 as an output and thus need the same type of value as inputs to training along with real-valued vectors. Support vector machines accept real-vectors with an associated class (integer). Association rule learning expects a set of binary attributes and a database of uniquely identified subsets of attributes.

The problem of automated software defect detection must ultimately translate source code, which can be thought of most simply as a stream of characters or more commonly as a tree or graph of control structures, data stores and associated information, into something a perceptron, a support vector machine or an association rule learner would understand. The alternative is to default to the current state of practice, which is to hand-code recognition algorithms that operate on trees or graphs for each unique case or to remain at the token processing level with tools such as regular expressions.

D. Classification

Given that the translation problem can be solved, many classification algorithms exist today that can be customized to the task of machining source code tokens, fragments or flows to defect classes. Different classifiers have different strengths and weakness that may fit a certain needs. For example, association rule learners have been applied to use of related function calls in source code. The learners find sets of functions that are used in proximity and report when one of the function calls frequently in the set do not exist in some instance[7]. One instance of a rule learned in this manner is for programming with the C++ standard library. An association can be that, with some probability, each time the *ifstream* token is seen the *open*, *read* and a *close* tokens are also seen. The following figure shows sample code where the *close* token is missing, which results in a leaked file handle.

```
void leaky(char const *name) {
    ...
    ifstream *ifs = new ifstream();
    ifs->open(name);
    while (! ifs->feof()) {
        c = read(...)
    }
}
```

Figure 2: Leaking a File Handle

A classification problem endemic to software defect detection is that it is typical to have orders of magnitude more non-defective lines of code or tokens than defective ones. Classifiers that measure their training success by reaching some accuracy percentage appear to have good results by classifying all tokens or lines as non-defective when one defect exists per hundred lines of code. However, such a classifier is of no utility. To be effective in software defect detection, a classifier must learn the relatively rare cases that are defects without producing so many false positives that a developer cannot see through the noise.

Jiang, Li and Zhou [8] claim to make improvements in dealing with imbalances in the number of defects with respect the volume of source code by implementing disagreement-based semi-supervised learning. This learning technique uses multiple learners that ingest both labeled and unlabeled data for training. When a majority of learners strongly agree on a classification for unlabeled data, the majority learners teach the minority learners using the examples on which they strongly agree.

E. Post-Processing

The preprocessing and feature extraction phases of the framework go to pains to ensure that the classification can produce results of interest, however, those results must be in a form usable by developers. That is, at minimum a developer would expect to see a result that identified a particular defect type identified for a particular source file at a particular line and perhaps column number. Additionally, some trace back of the reasoning behind calling the item a defect or at least a probability of being a defect should be reported. A simple output might look as can be seen in Figure 3 where defects are reported for two source files.

```
null-pointer, sourcefile1.c, 229
array-out-of-bounds, sourcefile1.c, 736
memory-leak, sourcefile2.c, 17
null-pointer, sourcefile2.c, 44
```

Figure 3: Automated Defect Detector Sample Output

A more user friendly output would annotate code within an integrated development environment (IDE) or produce a hypertext markup listing of source code linked to defect definitions.

III. Feature Extraction

Software source code can be represented in many forms. A most primitive representation is to think of source code as a stream of characters. This representation is the simplest to ingest but provides no syntactic, semantic or control and data flow information to the machine learner. A step up from character-by-character inputs is to work with lexical tokens as they appear in a source file. Further, lists of consecutive tokens can be constructed to serve as inputs, essentially giving a

learner a sliding window of consecutive tokens. Such an approach, while requiring lexical analysis remains far simpler than the complex parsing, semantic analysis and inter-procedural analysis necessary to hand-code automated source code analysis routines. Preliminary work by the researcher applied back propagation neural networks to streams of individual characters, lists of characters, individual lexical tokens, and list of lexical tokens with each character or token hashed into a unique integer for input into the neural network. The input characters/tokens were paired with an indication of defective (1) or non-defective (0) for each token. In the case of lists of tokens, anytime a list contained a character or token marked as defective, the list was marked defective. Otherwise the list was marked as non-defective. Figure 4 illustrates the source to hash translation.

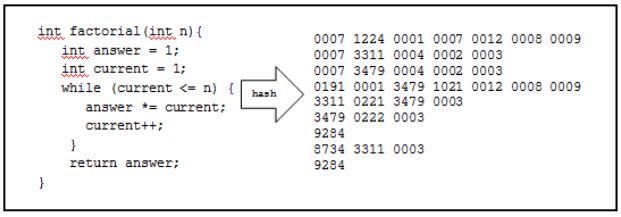

Figure 4: Source Code Hash Transformation

This simplistic approach was applied to a C++ code base on the order of tens of thousands lines of source code with nine expert-validated defects. Different list sizes were applied in the experiments and typical experimentation with different learning rates that is typical with neural networks was performed. While some defects could be reproduced by the learner, consistent results were not achieved. A number of limitations exist that prevent good performance in this case. With small list sizes, the learner has very little context. To learn to detect memory leaks with the context of a method, a learner would need a large window in order to see both the memory allocation and deallocation (or failure to deallocate) or else no viable learning could take place.

Another issue with this simple approach is that no semantic information is present nor is any generalization done. Suppose that a learner is being taught examples of off-by-one errors so non-defective examples might have conditions that look like *x* < *n* where defective examples may have conditions that look like *x* <= *n*. One instance might use the variables *x* and *n*, the next might use the variables *y* and *k* and another might use *a* and *b*. Each of the different variables has different representations to the learner and creating a correlation is difficult if not impossible. Additional preprocessing can help this situation if semantic information is included. For example, using semantic information and generalizing each variable into a pseudo-type like could transform each case into the form *int-var* < *int-var* or *int-var* <= *int-var*. Here a learner can find patterns more effectively regardless of differently named variables. The tradeoff here is that no learning can take place that depends on the data flow of a particular variable. Varied approaches will be necessary based on the problem. Fortunately, a human need not select from the possible set of approaches after they are defined since a learner can train using each of the possible approaches and select the approach that provides the best results.

This very lightweight approach, while ineffective in its first incarnation demonstrates that potential exists for transforming source code into a form that a machine learner can utilize. This contrasts with heavyweight approaches such as is found in [9] where the researchers modify the *gcc* compiler to perform interprocedural analysis that applies state machines hand-coded in a custom language to code samples to detect defects. Rather than using real instances of defects as examples or mutating known good code into defect samples, new types of defects require an error prone coding exercise in a unique programming language using a modified version of a compiler. Besides the complexity of hand-coding, this approach suffers from having to keep pace with the evolution of *gcc*.

While having to directly modify a compiler or write a new one is problematic, making use of the products of existing parsing technology makes an approach much more compatible with real-world source code and of less magnitude to deploy. For example, *gcc* produces various products such as abstract syntax trees, preprocessed code files, and assembler representations that can be preprocessed to serve as input to the feature extraction process.

One logical representation for source code is a tree or graph. Such structures map neatly to the syntactic structure or control and data flow of source code. Most machine learning algorithms cannot directly ingest a tree or graph, however, so a flattening or portioned approach to providing data to the learner must be taken. For example, walking a tree or graph via a depth first search, presenting scalar-valued nodes to the learner in sequence is a possible approach. The scalar-valued nodes may be formed by hashing tokens as previously described in this section.

IV. Classification

A wealth of publicly available classification engines are available in toolkits such as Weka[12]. Work has been performed at determining which apply well to software engineering data[14][15][16][17] and limitations of static analysis approaches and the difficulty in learning software defects have been reviewed[18]. Further, collections or ensembles of such standards learners have been formed to take advantage of the strengths and overcome the weaknesses of particular learners. Additionally, a comparison of approximate versus exhaustive application of such learners has been studied[19]. The vast body of research with these standard learners makes them a good starting place for application of automated defect detectors at the line of code level (rather than at the metric level) and the products of research in feature extraction should be exhausted before extensive work into customized classifiers is attempted.

One unique area is where there exist algorithms that directly compare graphs or trees[13], which cannot typically be ingested by typical machine learners. Graph comparison algorithms may be an avenue for future endeavors in software defect classification[10].

V. Building a Foundation

Hand-coded tools that address automated source code analysis frequently walk the execution paths of the software examining the possible states of variables for undesirable conditions such as dereferencing a null pointer or reference or writing past the end of an array. An approach using machine intelligence can model this approach of control and data flow analysis a number of ways. The current tactic in work for this research is to present to the machine learner a control flow graph. This graph is produced by requesting the *gcc* compiler produce an assembler code listing with full debug information. The *GNU Compiler Collection (GCC)* for which *gcc* is the driver program[11] is most useful because it accepts numerous different languages, all of which can be translated to assembler code. Providing the assembler level information to the learner makes it rather language neutral. The debug information allows the post-processing to map back from the assembler information to the high-level code that a developer will understand.

The classification approaches currently being investigated are two-fold. One is to feed a widely available machine learner such as a back propagation neural network sequences of control flow elements that walk the graph along with indications of the defect class associated with sequence. There are large variations that can be applied here. The length of the sequences and the particulars of what exactly is presented in each node can be varied as was discussed in Section III in addition to parameters that exist for the machine learners themselves. The second classification approach is to use the a measure of *graph edit distance* to compare subgraphs extracted for a training set to subgraphs that have a known defect or are known to be non-defective. Both approaches can be generalized by attempting to learn multiple classes of defects within a single knowledgebase or specialized such that a given learned knowledgebase only applies to a specific type of defect. Generalization provides versatility and potentially improved run-time performance. Specialization may provide improved correctness of the learner and shorter training cycles with fewer exemplars needed at the expense of run-time performance after training is complete (each specialized knowledgebase would need to be applied to the full code set).

Outputs of the intelligent defect detectors are being compared against expert validated, documented issues with operational software as well as commercial automated source code analysis tools. Further, variations in learning parameters and representations are being documented and compared. Results will be provided in a future publication.

VI. Looking Forward

As researchers continue to advance the use of machine learning technology in the field of software engineering, they must keep in mind that such technology is only valuable if it can perform the same function that could be done by hand either more efficiently or in a manner that produces higher accuracy (or other relevant metric). In particular, for automated software defect detection care must be taken that the preprocessing and training by example is not so cumbersome that it is no easier than hand crafting a solution by coding in a high-level language. Similarly, if the results of the learning produce so many false positives or miss so many true defects then interest may wane in applying these technologies to this domain.

With this in mind and knowing the numerous successes in applying machine learning to software metric ([20][21][22] are a few demonstrates of the reach of these techniques), researchers should look for opportunities to exploit other less than obvious metrics that may be of value. Some metrics that may produce correlations could be lexical distance between tokens or relative frequency of specific tokens in a compilation unit. Such simple metrics appear to have no relationship to the volume of code or the control flow of the code but such metrics are simple to compute and may have intrinsic value that none can predict.

Researchers must continue to leverage available tools for preprocessing and the wide variety of available learning engines. Key to linking these well-known pieces is the feature extraction “glue” that bridges the gap between source code in a text file and (often) numeric data provided to a machine learner.

Acknowledgment

The author would like to thank the management of the NASA GSFC Space Network, Ground Software Systems Branch, and White Sands Complex as well as his West Virginia University examining committee for their guidance in the relevance of this research and the resources to accomplish it.

References

[1] http://msdn.microsoft.com/en-us/library/3z0aeatx.aspx

[2] http://gcc.gnu.org/onlinedocs/gcc-4.5.2/gcc/Warning-Options.html#index-Wall-234

[3] http://download.oracle.com/javase/6/docs/technotes/tools/windows/javac.html

[4] http://www.mathworks.com/products/polyspaceclientc/description3.html

[5] http://www.klocwork.com/products/documentation/current/Detected_C_and_C%2B%2B_Issues

[6] M. Reformat, W. Pedrycz, and N. Pizzi, “Software Quality Analysis with the use of Computational Intelligence,” in Proc. 2002 IEEE Int’l Conf. Fuzzy Systems (FUZZ-IEEE’02), vol. 2, May 2002, pp. 1156-1161.

[7] Z. Li and Y. Zhou, “PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code,” in Proc. 2005 Joint 10th European Software Engineering Conf. (ESEC-FSE’05), September 2005, pp. 306-315.

[8] Y. Jiang, M. Li, and Z. Zhou, “Software defect detection with ROCUS,” Journal of Computer Science and Technology, in press.

[9] S. Hallem et al, “A System and Language for Building System-Specific, Static Analyses” in *Proc. Programming Language Design and Implementation* (SIGPLAN’02), ACM Press, 2002, pp. 69-82.

[10] S. Kim et al, “Automatic Identification of Bug-Introducing Changes,” *Proc. 21st Int’l Conf. On Automated Software Eng.* (ASE’06), IEEE CS Press, 2006, pp. 81-90.

[11] http://en.wikipedia.org/wiki/GNU_Compiler_Collection

[12] http://*www.cs.waikato.ac.nz/ml/weka/*

[13] M. Neuhaus and H. Bunke, "A Convolution Edit Kernel for Error-tolerant Graph Matching," in *Proc. 18th Int'l Conf. Pattern Recognition* (ICPR'06), IEEE CS Press, 2006, pp. 220-223.

[14] E. Ceylan, F. Kutlubay, and A. Bener, "Software Defect Identification Using Machine Learning Techniques," in *Proc. 32nd EUROMICRO Conf. Software Eng. and Adv. Applications* (EUROMICRO-SEAA'06), IEEE CS Press, 2006, pp. 240-247.

[15] G. Boetticher, "Nearest Neighbor Sampling for Better Defect Prediction," in *Proc. 2005 Wksp. Predictor Models Software Engineering* (PROMISE'05), ACM Press, 2005, pp. 1-6.

[16] T. Menzies et al, "Implications Of Ceiling Effects In Defect Predictors," in *Proc. 4th Intl. Workshop Predictor models in Software Eng.*, ACM Press, 2008, pp. 47-54.

[17] T. Menzies et al, "Assessing Predictors of Software Defects," in *Proc. 4th Int'l Workshop on Predictor Models in Software Eng.* (PROMISE'08), ACM Press, 2008, pp. 47-54.

[18] N. Fenton and M. Neil, "A Critique of Software Defect Prediction Models", in *IEEE Trans.on Sofware. Eng.*, IEEE CS Press, 1999, pp. 675-689.

[19] M. Benson, "Report to the Qualifying Exam Committee," unpublished

[20] S. Bibi et al, "Software Defect Prediction Using Regression via Classification," in *Proc. IEEE Int'l Conf. Computer Systems and Applications* (AICCSA'06), IEEE CS Press, 2006, pp. 330-336.

[21] T. Khoshgoftaar and R. Szabo, "Using Neural Networks to Predict Software Faults During Testing," in *IEEE Trans. Reliability*, vol. 45, Sep. 1996, pp 456-462.

[22] S. Heckman and L. Williams, "On Establishing a Benchmark for Evaluating Static Analysis Alert Prioritization and Classification Techniques," in *Proc. 2nd ACM-IEEE Int'l Symp. Empirical Software Eng. and Measurement* (ESEM'08), ACM Press, 2008, pp. 41-50.